Une architecture SMT pour le temps-réel strict

Jonathan Barre, Christine Rochange, Pascal Sainrat

Une architecture SMT pour le temps-réel strict

Jonathan Barre, Christine Rochange et Pascal Sainrat

Institut de Recherche en Informatique de Toulouse
Université de Toulouse - CNRS
31062 Toulouse cedex 9, France
{barre, rochange, sainrat}@irit.fr

Résumé
Les processeurs multi-flot simultané (Simultaneous Multithreading ou SMT) peuvent être de bons candidats pour satisfaire les exigences en performances toujours croissantes des applications embarquées. Toutefois, les architectures SMT classiques ne présentent pas toujours la prévisibilité temporelle nécessaire pour permettre une analyse statique de temps d'exécution pire cas (Worst-Case Execution Times ou WCET). Dans cet article, nous analysons la prévisibilité de différentes politiques de contrôle des ressources partagées implémentées sur les cœurs SMT existants. Ensuite, nous proposons une architecture SMT conçue pour exécuter un thread temps-réel strict de façon à ce que son temps d'exécution pire cas soit analysable même si d'autres threads (non critiques) sont exécutés simultanément. Des résultats expérimentaux montrent que cette architecture reste performante, en moyenne et dans le pire cas.

Mots-clés : temps-réel, architecture de processeur, temps d'exécution pire cas (WCET)

1. Introduction

Ces dernières années, la complexité des logiciels embarqués s'est accrue exponentiellement. Par exemple, on s'attend à ce que la prochaine génération de voitures haut de gamme n'inclue pas moins de 1 Go de code binaire [1]. L'explosion de la taille des logiciels est due à l'implémentation de fonctionnalités de plus en plus nombreuses, que ce soit pour améliorer la sécurité (ex : système d'antiblocage des freins ou ABS), pour augmenter le confort des usagers (ex : contrôle automatique des essuie-glaces) ou pour contribuer à la préservation de l'environnement (ex : contrôle de l'émission des gaz nocifs).

Dans le même temps, il n'est pas souhaitable d'augmenter le nombre de nœuds de calcul au même rythme car cela augmenterait la complexité des interconnexions. Une solution est alors d'intégrer plusieurs tâches sur un même nœud de calcul, comme préconisé par les initiatives AUTomotive Open System ARchitecture (AUTOSAR : `www.autosar.org`) et Integrated Modular Avionics (IMA : norme ARINC 651, `www.arinc.com`). Pour supporter l'exécution de plusieurs tâches sur un même nœud de calcul, il parait inévitable que les processeurs haute-performance, comme ceux qu'on trouve actuellement dans les ordinateurs de bureau, seront de plus en plus utilisés dans les systèmes embarqués. Nous pensons que les cœurs multiflot (Simultaneous MultiThreading ou SMT) pourraient être de bons candidats, en particulier quand des tâches de différents niveaux de criticité doivent être exécutées sur un même nœud.

Toutefois, les processeurs hautes performances mettent généralement à mal les exigences de prévisibilité des applications temps-réel strict. Même si des solutions existent pour estimer des WCETs de manière aussi précise que possible [12][16], le parallélisme de flot ajoute de nouvelles difficultés et nous pensons qu'elles ne peuvent pas être surmontées par les techniques statiques d'estimation de WCET, à moins que le matériel soit conçu pour la prévisibilité. Dans cet article, nous proposons une architecture multiflot adaptée au calcul de WCET. Cette architecture doit être capable d'exécuter *un*

thread ayant des contraintes temps-réel strictes en parallèle avec des threads moins critiques. Le comportement temporel du thread temps-réel strict doit être prévisible par analyse statique pour que l'on puisse prouver qu'il satisfait toujours ses échéances. La solution consiste à choisir/concevoir soigneusement les politiques de contrôle du partage des ressources entre les threads concurrents (distribution et ordonnancement).

Nous sommes conscients que de nombreuses applications pourraient nécessiter que plusieurs threads à contraintes temps-réel strictes soient exécutés sur le même nœud de calcul. Notre architecture n'assure la prévisibilité temporelle que pour *un seul* thread critique, mais elle peut être considérée comme un premier pas vers la conception d'une architecture supportant plusieurs threads critiques.

L'article est organisé comme suit. Dans la section 2, nous donnons quelques informations de base sur les architectures SMT. En particulier, nous passons en revue les stratégies de distribution de ressources et d'ordonnancement proposées dans la littérature et mises en œuvre dans les processeurs SMT du commerce. La section 3 décrit l'architecture que nous proposons pour exécuter un thread temps-réel strict de manière prévisible en parallèle avec des threads non critiques. Des résultats expérimentaux sont donnés et analysés dans la section 4. La section 5 fait un état des travaux associés et nous concluons l'article dans la section 6.

2. Multiflot simultané et prévisibilité temporelle

Les processeurs SMT exécutent plusieurs threads en même temps pour améliorer l'utilisation des ressources matérielles (principalement les unités fonctionnelles) [18]. Les threads concurrents partagent des ressources communes : files d'instructions, unités fonctionnelles, mais également les caches de données et d'instructions et les tables de prédiction de branchements. Dans cet article, nous nous concentrons sur les ressources du pipeline (nous laissons les problèmes liés aux caches et à la prédiction de branchement pour des travaux futurs).

Deux catégories de ressources du pipeline doivent être distinguées : les ressources de *stockage* (files d'instructions et tampons) gardent les instructions pendant un certain temps, généralement pour plusieurs cycles, tandis que les ressources de *bande passante* (ex : unités fonctionnelles ou étage de retrait) sont typiquement réallouées à chaque cycle [15]. Le partage des ressources de stockage est contrôlé à la fois en terme d'espace et de temps : il y a plusieurs politiques possibles pour distribuer les entrées de ressources parmi les threads actifs (politiques de *distribution*) et pour choisir les instructions qui quitteront la ressource à chaque cycle (politiques d'*ordonnancement*). Le partage spatial n'a pas de sens pour les ressources de bande passante car elles sont réallouées à chaque cycle. En fonction des stratégies de distribution et d'ordonnancement, le partage des ressources peut être une source majeure d'indéterminisme pour le comportement temporel d'un thread.

Dans cette section, nous décrivons les politiques de distribution et d'ordonnancement les plus courantes (que ce soit dans la recherche ou dans des projets industriels) et nous soulignons les problèmes de prévisibilité qui peuvent se poser quand on considère un thread temps-réel strict s'exécutant avec des threads arbitraires sur un processeur SMT.

2.1 Politiques de distribution de ressources

Le procédé le plus flexible pour distribuer les entrées d'une ressource (ex : une file d'instruction) entre les threads est la politique de *distribution dynamique* selon laquelle n'importe quelle instruction de n'importe quel thread peut prétendre à n'importe quelle entrée libre.

Cette politique a été la première considérée dans la recherche académique. Elle a été implicitement retenue pour maximiser l'utilisation des ressources (ce qui est l'objectif premier de l'exécution SMT), assurant que chaque ressource puisse être utilisée si un thread en a besoin. Néanmoins, comme il n'y a pas de limite au nombre d'entrées d'une ressource allouée à un thread, certains threads pourraient souffrir de famine quand toutes les entrées d'une ressource sont utilisées par les autres threads. Le moment où la famine peut arriver dépend des comportements respectifs des threads concurrents. Si on considère un thread temps-réel, on ne peut pas garantir qu'une de ses instructions nécessitant une entrée dans une ressource distribué dynamiquement l'obtiendra immédiatement, et le temps que cette instruction peut avoir à attendre avant d'être admise par la ressource ne peut pas être borné. Ainsi, avec une distribution dynamique, le WCET d'un thread ne peut pas être estimé à cause de la grande variabilité des délais d'accès à des ressources partagées. Cette politique n'est donc pas appropriée pour des applications temps-réel strict.

La politique de *distribution dynamique avec seuil* a été conçue pour minimiser les risques de famine. Maintenant, chaque thread ne peut occuper plus d'emplacements dans une ressource qu'un seuil fixé (habituellement un pourcentage de la capacité de la ressource, supérieur à 1 divisé par le nombre de threads). Un seul thread ne peut pas monopoliser toutes les entrées. Cette politique est utilisée pour contrôler les ordonnanceurs d'instructions du Pentium 4 à architecture Hyper-Threading [14].

Dans le contexte des applications temps-réel strict, la distribution dynamique avec seuil est également de nature non prévisible : on ne peut pas déterminer si un thread pourra avoir autant d'entrées de ressource que le seuil puisque certaines entrées peuvent être utilisées par d'autres threads. Aussi, le thread pourrait être retardé pour l'obtention d'une entrée dans une ressource de stockage, même s'il n'a pas atteint le seuil, et ce délai ne peut être borné. Ainsi, il n'est pas possible de dériver une estimation précise de WCET pour un thread temps-réel.

La distribution des ressources peut aussi être *statique* : chaque ressource est partitionnée et chaque thread a un accès privé à une des partitions. Cela empêche les famines et assure un accès équitable aux ressources communes pour tous les threads. Les performances ne sont peut-être pas optimales dans ce cas, car des threads peuvent être ralentis à cause d'un manque de ressource, tandis que d'autres threads sous-utilisent leur partition. Le partitionnement statique est largement utilisé pour gérer les files d'instructions dans les implémentations SMT [9][14], sauf dans les zones du pipeline où les instructions sont traitées dans le désordre (comme les files d'émission du Power5 et les files d'ordonnancement du Pentium 4). La raison en est qu'une file partitionnée est plus facile à implémenter qu'une file partagée dynamiquement.

La distribution statique des ressources est naturellement la plus adéquate pour un système temps réel strict car elle est totalement déterministe. En effet, chaque thread se voit accorder une part fixe des ressources qu'il ne peut dépasser et qui ne peut pas être utilisée par un autre thread. Ainsi le comportement d'un thread en ce qui concerne les ressources partagées statiquement ne dépend pas des threads environnants.

2.2. Politiques d'ordonnancement

En plus de leur politique de distribution, les ressources partagées sont aussi contrôlées par une politique d'ordonnancement qui joue le rôle d'arbitre entre les threads pour le choix des instructions qui peuvent quitter la ressource et avancer dans le pipeline. Nous allons passer en revue différentes politiques.

L'algorithme le plus courant est le très simple *Round-Robin* (RR). Chaque thread se voit accorder une opportunité à son tour, de manière circulaire, sans tenir compte du comportement des autres threads. Il faut distinguer le *round-robin optimisé* (O-RR), qui saute le tour d'un thread qui n'a pas d'instruction prête ou un thread non actif, du *round-robin strict* (S-RR) qui sélectionne éventuellement un thread inactif. Comme l'algorithme RR est totalement indépendant de ce qui se passe dans le processeur, il est réputé avoir des performances modérées, en contraste avec les politiques se souciant du contexte.

En ce qui concerne les applications temps-réel strict et le calcul du WCET, S-RR est une politique très prévisible : il est très facile de déterminer quand les instructions d'un thread seront choisies puisqu'on est sûr que le thread sera sélectionné tous les n cycles si le processeur est conçu pour gérer jusqu'à n threads actifs. L'algorithme O-RR altère légèrement la prévisibilité mais le délai entre deux sélections d'un thread peut toujours être borné.

L'accès de presque toutes les files aux ressources en aval est contrôlé par la politique O-RR dans le Pentium 4 [14] et par une stratégie S-RR sur le Power5 [9].

La politique *icount* est une autre stratégie possible souvent utilisée pour ordonnancer la file de lecture des instructions dans les projets académiques [2][3][5]. Chaque thread a une priorité dynamique qui est réévaluée à chaque cycle pour refléter le nombre de ses instructions présentes dans les étages pré-émission du pipeline (la plus haute priorité est donnée au thread qui a le moins d'instructions dans ces étages). Des instructions de plusieurs threads peuvent être lues en même temps, le nombre d'instructions lues par chaque thread étant fonction de sa priorité. Plusieurs variantes de la stratégie *icount* ont été proposées dans la littérature : elles utilisent divers compteurs comme base pour la mise à jour des priorités des threads. Par exemple, la politique *brcount* [19] considère le nombre de branchements dans les étages pré-émission alors que la politique *dcra* est fondée sur l'usage des ressources [5].

Malheureusement, un ordonnancement basé sur des priorités dynamiques de threads rend le comportement temporel d'un thread dépendant des autres threads actifs. Aussi, la politique *icount* et ses dérivées ne peuvent pas être utilisés dans le cadre du calcul de WCET pour un thread temps-réel strict.

Une autre politique, que nous qualifierons de *parallèle*, partitionne la bande passante entre les threads en sélectionnant le même nombre d'instructions pour chaque thread. Cela veut dire que tous les threads ont des instructions qui progressent à chaque cycle. A notre connaissance, cette politique n'est utilisée que dans le Power5 d'IBM pour sélectionner les instructions à retirer [9]. Chaque cœur de ce processeur peut supporter deux threads simultanés et peut retirer un groupe de cinq instructions pour chaque thread à chaque cycle. Cette politique est totalement déterministe car la progression d'un thread est indépendante des autres threads.

3. Une architecture SMT prévisible

Notre objectif est de concevoir une architecture SMT qui rend possible l'analyse du WCET des tâches temps-réel. Dans ce premier travail, nous proposons un processeur SMT où un thread temps-réel strict peut s'exécuter avec d'autres threads non critiques. Le thread temps-réel doit s'exécuter sans interférences avec les autres threads pour présenter un comportement temporel analysable. Dans le même temps, un certain niveau de performances doit être maintenu, au moins pour les autres threads, afin que les avantages de l'exécution SMT ne soient pas annulés par nos modifications de l'architecture.

3.1. Structure de base du Pipeline

Pour illustrer notre approche, considérons le pipeline de base présenté sur la Figure 1. Dans l'étage de lecture des instructions (IF), les instructions d'un thread sont lues dans le cache d'instructions et rangées dans la file de lecture des instructions (FQ). Là, elles attendent d'être sélectionnées pour le décodage (étage ID), après quoi elles entrent dans la file de décodage (DQ). Après le renommage (étage RN), elles sont rangées dans le tampon de ré-ordonnancement (ROB). L'étage EX (exécution) sélectionne à partir du ROB des instructions dont les opérandes sont prêts et pour lesquelles une unité fonctionnelle est disponible. Des instructions appartenant au même thread peuvent être exécutées *dans le désordre* pour améliorer le parallélisme d'instructions. Les instructions terminées sont finalement enlevées du ROB, dans l'ordre du programme, par l'étage CM (retrait) et quittent le pipeline.

3.2. Politique de distribution de ressources

Pour obtenir la prévisibilité temporelle d'un thread critique, chaque ressource de stockage (i.e. files d'instructions et de décodage, tampon de ré-ordonnancement) est partitionnée statiquement. Comme indiqué dans la section précédente, seul le partitionnement statique peut rendre le comportement temporel d'un thread temps-réel analysable. Ceci est illustré dans la Figure 1 où l'on considère un processeur pouvant supporter deux threads actifs : les files de lecture, de décodage et de ré-ordonnancement sont distribuées statiquement en deux partitions, une pour chaque thread.

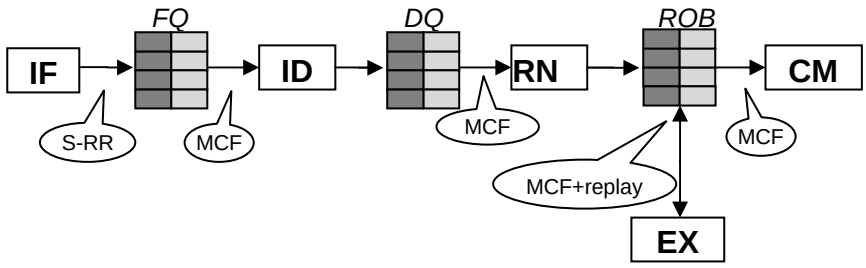

Fig. 1. Un pipeline SMT temporellement prédictible

3.3. Ordonnancement des threads

A chaque cycle, l'étage IF lit une séquence d'instructions pour un seul des threads, sélectionné par une politique S-RR dont nous avons dit qu'elle était prévisible.

Comme indiqué précédemment, la file de lecture est partitionnée statiquement. Nous devons maintenant spécifier l'algorithme implémenté pour sélectionner les instructions à décoder parmi celles présentes dans la file de lecture. Pour que le thread temps-réel strict (que nous désignerons par *hrt-t*, ou *hard real-time thread*, par la suite) avance de manière indépendante des threads concurrents, nous considérons un mécanisme à priorités partielles fixes, appelé *Most-Critical-First* (MCF), où les instructions du thread *hrt-t* sont sélectionnées en premier lieu. Si le nombre d'instructions du thread *hrt-t* dans la file de lecture est inférieur à la bande passante de décodage, des instructions des threads non critiques sont également sélectionnées selon un algorithme O-RR. Après le décodage, les instructions entrent dans la file de décodage qui est partitionnée. Elles sont sélectionnées pour le

renommage à l'aide de la stratégie prévisible MCF décrite ci-dessus. Une fois renommées, les instructions sont insérées dans le tampon de ré-ordonnancement partitionné statiquement où elles attendent leurs opérandes sources. Quand une instruction est prête, elle devient éligible pour l'envoi à une unité fonctionnelle. En plus de l'ordonnancement entre instructions d'un même thread (ex : instructions les plus vieilles sélectionnées en premier), la politique d'ordonnancement des threads doit aussi choisir entre les threads qui ont des instructions prêtes. En ce qui concerne le thread *hrt-t*, le temps d'attente d'une de ses instructions prêtes ne doit pas dépendre des threads concurrents. Pour cela, on complète la politique MCF par un mécanisme que nous appelons *Replay*. Quand une instruction du thread *hrt-t* est prête à s'exécuter, trois situations peuvent être observées :

- si l'unité fonctionnelle est libre, l'instruction peut y être envoyée immédiatement ;
- si l'unité fonctionnelle requise est déjà allouée à une instruction appartenant également au thread *hrt-t*, l'exécution est retardée. Néanmoins, cette situation reste analysable statiquement car elle ne dépend que du comportement propre du thread.
- si l'unité demandée est utilisée par une instruction d'un thread non critique, cette instruction est éjectée de l'unité fonctionnelle et marquée prête dans le tampon de ré-ordonnancement. Cela signifie qu'elle devra être réémise plus tard. L'unité fonctionnelle est alors immédiatement attribuée au thread *hrt-t*. Grâce à l'ordonnancement MCF, cette situation ne peut arriver que lorsqu'un thread non critique exécute une opération à latence supérieure à un cycle sur une unité fonctionnelle non pipelinée (dans notre exemple de cœur, cela ne concerne que les divisions).

Finalement, les instructions terminées sont sélectionnées pour le retrait par le mécanisme MCF qui assure prévisibilité temporelle au thread temps-réel strict.

4. Evaluation des performances

4.1. Méthodologie de simulation

Nous avons mené les expérimentations rapportées dans ce papier à l'aide d'un simulateur de niveau cycle développé dans le cadre de la plateforme OTAWA [6]. OTAWA est destiné au calcul de WCET et comprend de nombreux utilitaires dont un simulateur de niveau cycle construit sur SystemC. Le simulateur modélise des architectures génériques de processeurs où chaque étage de pipeline est vu comme un composant qui lit des instructions à partir d'une file d'entrée, les traite en appliquant un ensemble d'opérations prédéfinies, et les insère dans une file de sortie. Chaque file est paramétrée par une politique de distribution et une politique d'ordonnancement. Ce simulateur temporel est piloté par un simulateur fonctionnel généré automatiquement à partir de la description du jeu d'instruction PowerPC grâce à notre outil GLISS [21].

4.2. Configuration du processeur

Pour nos expérimentations, nous avons dérivé notre architecture SMT prévisible d'un cœur superscalaire de degré 4 (i.e. chaque étage a une largeur de 4 instructions). Le nombre de threads simultanés a été fixé à 2 ou 4.

Nous avons considéré un prédicteur de branchements parfait (oracle) et un cache parfait (tous les accès sont des succès). Bien que nous n'ayons pas modélisé les caches (et en particulier les interférences entre threads à ce niveau), nous avons considéré un taux d'échec aléatoire de 1%, avec une latence de 100 cycles en cas d'échec.

Le Tableau 1 donne les principales caractéristiques du pipeline. Les files sont partitionnées statiquement en 2 ou 4 (suivant le nombre de threads). Dans nos expérimentations, nous considérons deux architectures : (a) le cœur *de base*, qui implémente un ordonnancement de type O-RR pour toutes les ressources excepté pour l'envoi aux unités fonctionnelles où les instructions les plus vieilles sont sélectionnées d'abord, quel que soit le thread auquel elles appartiennent ; (b) l'architecture *adaptée au WCET* que nous avons conçue pour être prévisible temporellement et qui implémente notre politique MCF ainsi que le mécanisme de Replay.

Paramètre		**Valeur**
Largeur du Pipeline		4
Taille de la file de Lecture (Fetch)		16
Taille de la file de Décodage		16
Taille du ROB		64
Latence des unités fonctionnelles	MEM (pipelinée)	2
	ALU1	1
	ALU2	1
	FALU (pipelinée)	3
	MUL (pipelinée)	6
	DIV	15

Tableau 1. Configuration de base

4.3. Programmes de test

Un ensemble de tâches simultanées est composé de 2 ou 4 threads compilés dans un même code binaire (ceci parce que notre simulateur ne peut gérer qu'un seul espace d'adressage). Le code source de chaque thread est encapsulé dans un appel de fonction et le compteur de programme correspondant est initialisé au point d'entrée de la fonction. Pour maximiser les interférences possibles entre threads (et donc nous situer dans un contexte défavorable), nous avons considéré des threads concurrents exécutant la même fonction. Les différentes fonctions utilisées au cours des tests sont listées dans le Tableau 2 (leur code source provient de la suite SNU-RT [22], une collection de tâches relativement simples exécutées couramment sur des systèmes embarqués temps-réel strict).

Fonction	**Description**
fir	filtre FIR avec générateur de nombres gaussiens
ludcmp	décomposition LU
lms	Amélioration adaptative de signal LMS
fft1k	FFT (transformée de Fourier) sur des tableaux de 1000 nombres complexes

Tableau 2. Fonctions de test

Tous les exécutables ont été compilés avec l'option `-O` qui enlève la plupart des accès inutiles à la mémoire tout en conservant la structure algorithmique du code (ce qui permet d'effectuer l'analyse de flot sur le code source).

4.4. Résultats expérimentaux

Pour servir de référence, nous avons simulé chaque thread en parallèle avec des threads fantômes, c'est-à-dire des threads qui n'exécutent aucune instruction et n'allouent aucune ressource partagée dynamiquement, et qui, ainsi, n'interfèrent en rien avec le thread principal. Toutes les ressources de stockage sont considérées partitionnées. Les résultats bruts (temps d'exécution du thread principal, en cycles, mesuré sur l'architecture de base non prévisible) sont donnés dans le Tableau 3. Le temps d'exécution d'un thread de référence est toujours plus long avec 4 threads qu'avec 2 car la capacité totale des ressources est la même dans les deux cas. Ainsi, dans la configuration avec 4 threads, chaque thread a des partitions privées plus petites.

	Nombre de threads	
	2	**4**
fir	25 022	47 897
ludcmp	4 165	8 205
lms	390 815	752 858
fft1k	1 239 147	2 384 030

Tableau 3. Temps d'exécution de référence

Le tableau 4 montre comment les mêmes threads s'exécutent sur l'architecture SMT de base (non prévisible) quand ils sont concurrents à d'autres threads (exécutant la même fonction). Les temps d'exécution rapportés sont ceux des derniers threads terminés : tous les autres threads ont un temps d'exécution moindre. Dans chaque cas, nous indiquons l'augmentation du temps d'exécution par rapport aux mesures rapportées dans le Tableau 3. Comme prévu, le temps global d'exécution pour 2 ou 4 threads « réels » est plus grand que le temps d'exécution d'un thread en parallèle avec des threads fantômes. Cela est dû à la compétition entre les threads pour l'accès aux ressources partagées dynamiquement. Le coût en performance est d'autant plus important que le nombre de threads actifs est grand.

Le Tableau 5 donne un aperçu de la fréquence et de la durée des délais subis par les instructions d'un thread à cause de celles d'autres threads. Les nombres indiquent le pourcentage d'instructions qui ont été retardées de n cycles pour obtenir une ressource (unité fonctionnelle). On voit qu'à peu près 10% des instructions sont retardées par un thread concurrent dans un cœur à 2 threads, dans un cœur à 4 threads ce taux est d'environ 5%. On peut noter que certaines instructions peuvent subir de sérieux retards (des délais allant jusqu'à 90 cycles et plus ont été observés). On remarque aussi que les instructions sont moins souvent retardées dans un cœur à 4 threads que dans un cœur à 2 threads. Ceci est dû à la taille des files d'instructions (2 fois plus petites pour un thread) : il y a ainsi moins d'instructions en attente et donc moins de conflits et de délais.

	Nombre de threads	
	2	**4**
fir	26 548 +6.1%	60 691 +26.7%
ludcmp	5 068 +21.6%	10 686 +30.2%
lms	420 161 +7.5%	951 210 +26.3%
fft1k	1 345 598 +8.6%	3 009 002 +26.2%

Tableau 4. Temps d'exécution dans le cœur SMT non prévisible

		Délais (nombre de cycles)					
		0	**1-5**	**6-10**	**11-20**	**21-30**	**>30**
fir	2 threads	92.93%	5.88%	0.76%	0.38%	0.02%	0.03%
	4 threads	96.66%	3.16%	0.09%	0.09%	0.00%	0.00%
ludcmp	2 threads	89.32%	9.83%	0.19%	0.43%	0.04%	0.19%
	4 threads	98.88%	1.01%	0.08%	0.04%	0.00%	0.00%
lms	2 threads	92.16%	7.08%	0.47%	0.24%	0.03%	0.03%
	4 threads	96.91%	2.89%	0.09%	0.11%	0.00%	0.00%
fft1k	2 threads	93.93%	4.18%	1.25%	0.58%	0.04%	0.02%
	4 threads	94.50%	5.29%	0.09%	0.11%	0.00%	0.00%

Tableau 5. Délais dûs aux threads concurrents (taux d'instructions)

Les résultats obtenus avec notre cœur prévisible (avec la politique MCF et le mécanisme Replay) sont donnés dans le Tableau 6. Encore une fois, les temps d'exécution sont ceux du dernier thread terminé. Ils ne concernent pas le thread temps réel strict qui a, *dans tous les cas*, le même temps d'exécution que celui donné dans le Tableau 3 (puisque le matériel fait en sorte qu'il ne soit pas perturbé par les autres threads). La perte de performance (en termes d'augmentation du temps d'exécution) par rapport à l'architecture SMT de base est indiquée dans chaque cas.

	Nombre de threads	
	2	**4**
fir	37 401 +40.9%	68 023 +12.1%
ludcmp	5 302 +4.6%	12 012 +12.4%
lms	520 369 +23.8%	1 001 526 +5.3%
fft1k	1 867 441 +38.8%	3 503 297 +16.4%

Tableau 6. Temps d'exécution dans le cœur SMT prévisible

Naturellement, les mécanismes d'ordonnancement que nous avons implémentés pour assurer la prévisibilité temporelle pour un thread critique dégrade les performances. Néanmoins la perte est modérée : 27% en moyenne pour 2 threads et 11,6% pour 4 threads. La raison pour laquelle cette perte est plus importante pour 2 threads que pour 4 est le coût du partitionnement statique des files d'instructions (le thread critique disposant de la moitié, au lieu d'un quart des ressources partitionnées). Ainsi, environ 25% (au lieu de 50%) des instructions sont traitées en priorité aux dépens des autres 75% (50%). Cela laisse plus d'opportunités aux threads non critiques pour s'exécuter normalement.

5. Travaux associés

Crowley et Baer [7] reprennent une approche largement utilisée pour le calcul du WCET (la technique IPET [10]) pour l'analyse d'un processeur SMT. Ceci conduit à exprimer tous les entrelacements possibles entre les threads dans le cadre de la formulation ILP (programmation linéaire en nombres entiers) du calcul de WCET. Naturellement, la taille du système ILP généré croit exponentiellement avec la taille des threads et, à moins de considérer des tâches très simples (avec peu de contrôle de flot), le problème ne peut être résolu en un temps raisonnable. Cette constatation est la source de notre travail sur les architectures SMT prévisibles : nous pensons que l'entrelacement des threads doit être contrôlé à l'exécution (i.e. par le matériel) pour le rendre efficace et déterministe afin qu'il puisse être pris en compte pour l'estimation du WCET de tâches ayant des échéances strictes.

D'autres travaux ont pour but de rendre le comportement temporel des threads plus prévisible à travers des stratégies appropriées d'ordonnancement au niveau système. Lo *et al.* [13] explorent des algorithmes d'ordonnancement de tâches temps-réel sur une architecture SMT. Toutefois, ils ne prennent pas en compte les interférences possibles entre les threads à l'intérieur du pipeline et leurs effets sur le WCET des threads. Dans [10], Kato *et al.* introduisent la notion de temps d'exécution multi-cas (*Multi-Case Execution Time* ou MCET), calculé à partir des différents WCETs du thread considéré quand il s'exécute en présence de différents threads concurrents. Nous pensons que le nombre de valeurs possibles du WCET peut être considérable dès que les threads concurrents ont un grand nombre de chemins d'exécution possibles (le nombre d'entrelacements peut alors être énorme). C'est pourquoi nous croyons que ces résultats requièrent une architecture prévisible (comme celle que nous proposons dans cet article) pour être applicables.

Dans [8], le but est de préserver autant que possible les performances de certains threads prioritaires donnés, tout en permettant aux autres threads de progresser. L'objectif est proche du nôtre sauf qu'il n'assure pas de prévisibilité temporelle et n'est ainsi pas approprié à un contexte temps-réel strict. Des mécanismes architecturaux ont été proposés par Carzola *et al.* [2][3][4] pour garantir une qualité de service pour un ensemble de threads. Cette solution cible principalement les systèmes temps-réels souples et dépasse le cadre de notre travail.

Le processeur CarCore [20] met en œuvre une exécution multiflot SMT avec deux pipelines spécialisés : un pour les instructions de calcul et un pour les accès mémoire. Des instructions de quatre threads actifs peuvent avancer en parallèle dans les pipelines. Cette architecture peut supporter *un* thread critique à l'aide d'une politique d'ordonnancement d'instructions basée sur des priorités. Contrairement à notre architecture, le processeur CarCore ne peut pas exécuter des instructions dans le désordre.

6. Conclusion

Des besoins de plus en plus importants, liés à une demande toujours plus grandissante de nouvelles fonctionnalités, rendront inévitable l'utilisation de cœurs de calculs avancés dans les systèmes embarqués dans un futur proche. Les cœurs multiflot semblent être de bons candidats pour exécuter plusieurs tâches de différentes criticités en parallèle. Toutefois, les processeurs SMT actuels ne présentent pas assez de prévisibilité temporelle pour satisfaire les exigences de certification des applications critiques. En fait, nous pensons que l'entrelacement des threads ne peut être traité par les techniques habituelles d'analyse statique. C'est pourquoi nous considérons que la solution réside dans la conception de matériel spécifique.

Nous avons proposé une architecture SMT prévisible dans laquelle les politiques implémentées pour contrôler le partage des ressources internes entre les threads sont conçues pour permettre l'exécution prévisible d'un seul thread temps-réel strict en concurrence avec d'autres threads moins critiques. Toutes les ressources de stockage (files d'instructions et tampons) sont partitionnées statiquement et l'ordonnancement des instructions fait appel à une stratégie *Most-Critical-First* qui donne priorité au thread temps-réel strict. On ajoute à ceci un mécanisme de *Replay* qui assure qu'une instruction du thread critique ne peut pas être retardée par une instruction d'un thread non critique lors de l'accès à une unité fonctionnelle.

Les résultats expérimentaux montrent que, alors que le thread temps-réel strict s'exécute aussi vite que s'il était seul dans le pipeline, la perte de performance pour les autres threads est modérée. Elle est inférieure à 12% pour un cœur SMT prévisible à 4 threads.

Pour poursuivre ce travail, nous avons l'intention de résoudre certains problèmes ignorés dans cette étude préliminaire, comme la stratégie de partage des caches de données et d'instructions, et du prédicteur de branchement, l'objectif étant toujours d'assurer une prévisibilité temporelle complète pour le thread critique. Nous comptons également développer des solutions permettant l'exécution simultanée de *plusieurs* threads critiques.

Références

1. M. Broy, I. Krüger, A. Pretschner, C. Salzmann, "Engineering Automotive Software", Proceedings of the IEEE, 95(2), 2007.

2. F. Cazorla, A. Ramirez, M. Valero, P. Knijnenburg, R. Sakellariou, E. Fernández, "QoS for High-Performance SMT Processors in Embedded Systems", IEEE Micro, 24(4), 2004.
3. F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, M. Valero, "Predictable Performance in SMT Processors", ACM Conf. on Computing Frontiers, 2004.
4. F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, M. Valero, "Architectural Support for Real-Time Task Scheduling in SMT Processors", Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2005
5. F. Cazorla, A. Ramirez, M. Valero, E. Fernández, "Dynamically Controlled Resource Allocation in SMT Processors", 37th Int'l Symposium on Microarchitecture, 2004.
6. H. Cassé, P. Sainrat, "OTAWA, a framework for experimenting WCET computations", 3rd European Congress on Embedded Real-Time Software, 2006.
7. P. Crowley, J.-L. Baer, "Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors", HPCA-9 Workshop on Network Processors, 2003.
8. G. Dorai, D. Yeung, S. Choi, "Optimizing SMT Processors for High Single-Thread Performance", Journal of Instruction-Level Parallelism, vol. 5, 2003.
9. R. Kalla, B. Sinharoy, J. Tendler, "IBM Power5 Chip: A Dual-Core Multithreaded Processor", IEEE Micro, 24(2), 2004.
10. S. Kato, H. Kobayashi, N. Yamasaki, "U-Link: Bounding Execution Time of Real-Time Tasks with Multi-Case Execution Time on SMT Processors", 11th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications, 2005.
11. Y.-T. Li, S. Malik, "Performance Analysis of Embedded Software using Implicit Path Enumeration", Workshop on Languages, Compilers, and Tools for Real-time Systems, 1995.
12. X. Li, A. Roychoudhury, T. Mitra, "Modeling out-of-order processors for WCET analysis", Real-Time Systems, 34(3), 2006.
13. S.-W. Lo, K.-Y. Lam, T.-W. Kuo, "Real-time task scheduling for SMT systems", 11th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications, 2005.
14. D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture", Intel Technology Journal, 6(1), 2002.
15. S. Raasch, S. Reinhardt, "The Impact of Resource Partitioning on SMT Processors", 12th Int'l Conf. on Parallel Architectures and Compilation Techniques, 2003.
16. C. Rochange, P. Sainrat, "A Context-Parameterized Model for Static Analysis of Execution Times", Transactions on High-Performance Embedded Architectures and Compilers, 2(3), 2007.
17. N. Tuck, D. Tullsen, "Initial observations of the simultaneous multithreading Pentium 4 processor", 12th Int'l Conf. Parallel Architectures and Compilation Techniques, 2003.
18. D. Tullsen, S. Eggers and H. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism". In 22nd Int'l Symposium on Computer Architecture, 1995.
19. D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", 23rd Int'l Symposium on Computer Architecture, 1996.
20. S. Uhrig, S. Maier, and T. Ungerer, "Toward a processor core for real-time capable autonomic systems", IEEE Int'l Symposium on Signal Processing and Information Technology, 2005.
21. www.irit.fr/Gliss
22. archi.snu.ac.kr/realtime/benchmark/